Distributed and collaborative software analysis

Ghezzi, G ; Gall, H C

Abstract: Throughout the years software engineers have come up with a myriad of specialized tools and techniques that focus on a certain type of software analysis such as source code analysis, duplication analysis, co-change analysis, bug prediction, or detection of bug fixing patterns. However, easy and straight forward synergies between these analyses and tools rarely exist because of their stand-alone nature, their platform dependence, their different input and output formats and the variety of data to analyze. As a consequence, distributed and collaborative software analysis scenarios and in particular interoperability are severely limited. We describe a distributed and collaborative software analysis platform that allows for a seamless interoperability of software analysis tools across platform, geographical and organizational boundaries. We realize software analysis tools as services that can be accessed and composed over the Internet. These distributed analysis services shall be widely accessible in our incrementally augmented Software Analysis Broker where organizations and tool providers can register and share their tools. To allow (semi)-automatic use and composition of these tools, they are classified and mapped into a software analysis taxonomy and adhere to specific meta-models and ontologies for their category of analysis.

DOI: https://doi.org/10.1007/978-3-642-10294-3_12

Distributed and Collaborative Software Analysis

Giacomo Ghezzi and Harald C. Gall

s.e.a.l. – software evolution and architecture lab
Department of Informatics, University of Zurich, Switzerland
{ghezzi,gall}@ifi.uzh.ch
http://seal.ifi.uzh.ch/

Summary. Throughout the years software engineers have come up with a myriad of specialized tools and techniques that focus on a certain type of software analysis such as source code analysis, duplication analysis, co-change analysis, bug prediction, or detection of bug fixing patterns. However, easy and straight forward synergies between these analyses and tools rarely exist because of their stand-alone nature, their platform dependence, their different input and output formats and the variety of data to analyze. As a consequence, distributed and collaborative software analysis scenarios and in particular interoperability are severely limited. We describe a distributed and collaborative software analysis platform that allows for a seamless interoperability of software analysis tools across platform, geographical and organizational boundaries. We realize software analysis tools as services that can be accessed and composed over the Internet. These distributed analysis services shall be widely accessible in our incrementally augmented *Software Analysis Broker* where organizations and tool providers can register and share their tools. To allow (semi)-automatic use and composition of these tools, they are classified and mapped into a software analysis taxonomy and adhere to specific meta-models and ontologies for their category of analysis.

1 Introduction

A common feature of many software analysis tools is that they focus on just a particular kind of analysis to produce the results wanted by the engineer. If different analyses are required, an engineer needs to run several tools, each one specialized on a particular aspect, ranging from pure source code analysis, duplication analysis, co-change analysis, bug prediction, to bug fixing patterns and visualization. All these techniques have their own explicit or implicit meta-model which dictates how to represent the input and the output data. Thus the sharing of information between tools is only possible by means of a cumbersome export towards files complying to a specified exchange format. Also, if there exist several analyses of the same kind (*e.g.,* code duplication

analysis) there is hardly any way to compare the results or integrate them other than manual investigation. Tool interoperability is hampered even more by their stand-alone nature as well as their platform and language dependence.

As a consequence, distributed and collaborative software analysis scenarios are severely limited. This significantly restrains the usage and reduces the acceptance of software evolution analysis tools both by tool builders that would otherwise greatly benefit from that huge amount of diverse information. We claim that this status quo severely hampers software evolution.

1.1 Tools and IDEs

Lately, many software companies have been putting a lot of effort on tool integration to keep track and collect data on software development projects to enable and promote seamless collaboration on all the development phases. The main goal is to create a powerful and successful software development team collaboration platform to integrate work across the phases of the development life-cycle done by different actors. Examples for such IDEs are *IBM's Jazz*[1] or the upcoming *Microsoft's Visual Studio Team System 2010*[2]. Among the many functionalities, they fully integrate work item management with source control, team processes, build and test case management. For example, for a code change set information is provided why it was made (the associated work items), when it had some test problems, when it finally made it into a release, who has been working on it, or what source code changes were involved. But software analysis, and in particular release histories, are being left out of the picture. Thereby, these IDEs gather a huge amount of data on the development process but only a very little portion of it is then effectively used for analysis purpose.

From a research perspective, throughout the years, we have developed several tools to extract and analyze different types of data about a software project: its CVS release history and Bugzilla data [1], its FAMIX model [2], its fine-grained source code changes [3], its change types and couplings [4] and its developer networks [5]. All these tools are integrated into our software evolution platform called *Evolizer* which allows them to communicate and share their data. But what if we want to add an external, already existing analysis, say a code clone detector? Not only we would have to deal with language and platform dependency issues but even more, we would have to take care of inter-domain integration of the data produced by the new tool and the one already shared in our platform.

We argue that these challenges can be solved by means of software orientation. In this chapter we present our approach towards a *a distributed and collaborative software analysis platform that allows for interoperability of different software analyses across platform, geographical and organizational*

[1] http://jazz.net
[2] http://msdn.microsoft.com/en-us/vstudio/bb725993.aspx

boundaries. Particular analyses are represented in a software analysis taxonomy and adhere to specific meta-models and ontologies for their category. They offer a common web service interface that enables their composite use on the Internet. These distributed analysis services are accessible through an incrementally augmented *Software Analysis Broker*, where organizations and tool builders can register and share their analyses.

Allowing disparate analyses to be available as web services and interoperate by sharing their data would be highly beneficial for three reasons: (1) it can speed up the collaboration of software engineers by being able to share their analyses and use each others analyses with only little overhead; (2) not only tool builders but also analyses itself could collaborate as web services and they could be composed into chains of services or into more complex services with the web service composition language such as BPEL4WS [6]; and (3) it would facilitate the uncovering of new meaningful analyses based on a *Software Analysis Broker*.

1.2 A Scenario for Collaborative Software Analysis across Organizational and Tool Boundaries

Before going deeper into detailing our work, we briefly illustrate the challenges we want to address and the potential impact of our work with the following software analysis collaboration scenario:

Alice has developed a tool extracting the detailed CVS history of software projects to gain better insights on the development process. Bob has a tool doing the same but with Bugzilla data, and Charlie's tool extracts the Famix model of a given object-oriented software project by parsing its entire source code to obtain an unambiguous and precise language independent representation of it. Each of the tools works on a specific platform and requires its own settings and parameters. Alice, Bob, and Charlie do not work for the same institution. They decided to unify their efforts to throughly analyze the history of Foozilla, a multi-million lines of code system, but the communication overhead due to different data-models, different result data formats, and storage media are too cumbersome to follow-up on their exciting plan. So they start thinking of a unified software analysis platform that would allow them to easily get a detailed holistic view of the history of Foozilla: it would tell them for example for each release which bugs are related to specific files revisions, thus providing a clear link between a bug and some specific source code files. Bug prone parts, bug fixing and other source code change patterns would then be easy to spot. Based on this, new and additional analyses could be produced and offered on the same platform. For example, Jane could then develop the analysis she always wanted to implement but lacked the right expertise and tool support. That analysis calculates source code metrics (through its Famix model, without thus having to deal with the actual source code) for each CVS release to spot code smells to both show their trend over time and their relation

to reported bugs and eventually show that into some nice navigable graphical interface.

2 State-of-the-Art in Software Analyses

Software analysis is one of the key activities in software evolution as it allows to extract the most diverse and extensive information regarding a software system. The classic analyses have been around for years targeting models and source code [7]. In the last years many research groups have shifted their attention to software evolution and the whole established community of reverse engineering, reengineering, and program understanding has actually acknowledged that evolution is indeed the umbrella of their research activities.

There is a plethora of research on software analysis, but it is not in our intention to give a complete picture of the state of the art. We just want to sketch the type and range of analyses that can be integrated in our proposed service platform. In this way we want to better contextualize our approach and show its potential.

Approaches focusing on the software evolution either study its source code change history, bug history, its underlying dynamics or a combination of them. Fischer *et al.* [1] populated a release history database, combining information from version control and bug tracking systems, namely CVS and Bugzilla to facilitate further analysis. Draheim *et al.* [8] had a similar approach but only worked with version control data from CVS. Many other works detect and track changes made on the source code during the software project lifetime. Zou *et al.* [9] used origin analysis to detect merging and splitting while S. Kim *et al.* [10] used it to track function name changes. M. Kim *et al.* [11] focused just on code clone evolution and built a clone genealogy tool to extract code clones history from a project CVS repository.

Works by Zimmermann *et al.* [12] and Ying *et al.* [13] predict future source code changes given past source revision history of a project stored into CVS repositories to then recommend potentially relevant source code for a particular modification task. Source revision history is analyzed to extract also other kinds of information. Livshits *et al.* [14] combine that with dynamic analysis techniques to identify application-specific patterns and find pattern violation. Hipikat [15] forms an implicit group memory combining CVS source repository data, Bugzilla data, messages posted on developer forums and other project documents to recommend artifacts that are relevant to a particular task that a developer is trying to perform.

Gall *et al.* [16] extracted change couplings of software modules by analyzing CVS data, in particular check in and check out time and the authors of those actions; from that they were able to discover design flaws without analyzing a single line of code. Fluri *et al.* [4] focused on the extraction of several fine-grained source code change types and the assessment of their significance in terms of their impact on other source code entities and whether

they may be functionality-modifying or functionality-preserving. Then, Nagappan *et al.* [17] predicted defect density for a system using code churn metrics fetched from its change history.

Similarly to the works on source code change, bug analysis addressed extraction of data from a bug repository (as we already saw in [1]), its prediction or its analysis. For that, Hassan *et al.* [18] developed a dynamic cache of the ten mostly error prone subsystems (directories). Kim *et al.* [19] proposed a similar approach, but they dynamically cached the most likely fault prone source code locations. Sliwerski *et al.* [20] related version history and a bug database to detect, as Kim *et al.* [19], code locations whose changes had been risky in the past and annotated them with color bars to show their risk rate in Eclipse [21]. While much effort has been spent on software cost/effort prediction, very little has been done on bug fixing effort prediction. As for example the work by Weiss *et al.* [22] in which, for every new bug report in a issue tracking system, similar earlier reports are fetched and their average time is used as a prediction for the new one.

Not only the history of a software development process has been addressed, but also its underlying dynamics. In particular, a lot of research has also been performed on the role of the developers in evolutionary processes. For example, Čubranić *et al.* [23] and Anvik *et al.* [24] both developed approaches for bug triaging that recommend a list of developers with the appropriate expertise to solve a particular bug by applying machine learning techniques on bug reports fetched from a bug repository (in these cases Bugzilla). Mockus *et al.* [25] located people with desired expertise not using bug reports but by analyzing data from change management systems. Girba *et al.* [26] analyzed CVS logs to reconstruct code ownership to help in answering which authors are knowledgeable in which part of the system and also reveal behavioral patterns: when and how different developers interact in which way and in which part of the system.

The use of web services and ontologies for software analysis and evolution has been addressed only recently in research. A few works have used software analysis data and concept representations with ontologies. Hyland-Wood *et al.* [27] presented an OWL ontology of software engineering concepts including classes, tests, metrics and requirements. Happel *et al.* [28] in their KOntoR approach stored and queried meta-data about software artifacts to foster software reuse. What is interesting for us is that they proposed various ontologies to provide background knowledge about software components, such as the programming language and licensing models.

Highly related to our approach is the work by Kiefer *et al.* [29], which proposed EvoOnt, a software repository data exchange format including software, release and bug related information based on OWL. To effectively mine software systems represented in that OWL format and find, for example, code smells, they introduced iSPARQL, a query engine supporting similarity joins. From their work we borrow the idea of using ontologies to represent software analysis data to facilitate data exchange and automatic reasoning.

Jin and Cordy [30], with their Ontological Adaptive Service-Sharing Integration System (OASIS), are the first and so far only researchers that studied an ontology based software analysis tool integration system that employs a domain ontology and specifically constructed external tool adapters. They also implemented a proof of concept with three reverse engineering tools that allowed them to explore service-sharing as a viable means for facilitating interoperability among tools.

3 The Software Service Platform

There is a huge variety of tools and techniques out there offering the most disparate analyses on a software system, but it is impossible for researchers and software companies to easily and effectively share, combine and integrate them. What follows is the description of how we tackle the problem.

3.1 The Software Analysis Broker Infrastructure

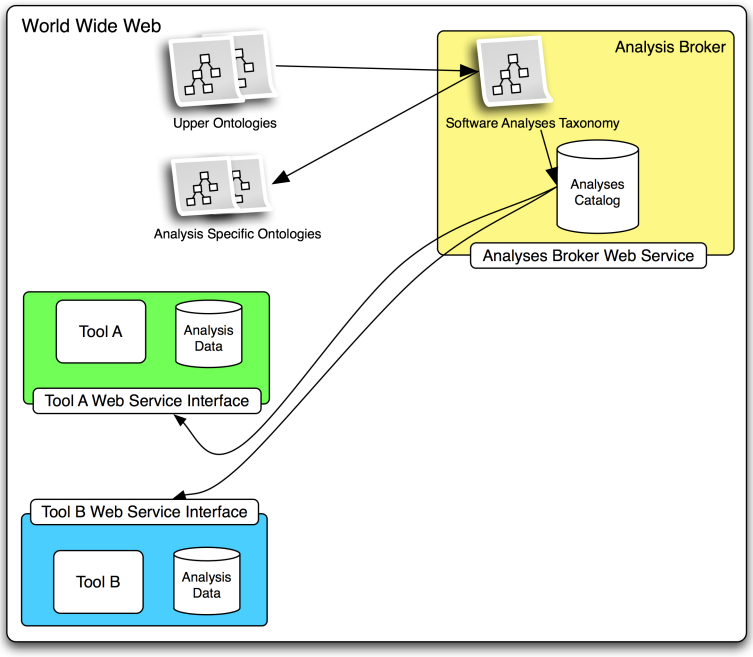

Fig. 1: Overview of our software analysis service platform

Figure 1 gives an overview of our approach, which is made up by four main constituents: software analysis web services, an analysis services catalog,

a *Software Analysis Broker* and ontologies. Software analysis web services "wrap" already existing analysis tools exposing their functionalities and data through a web service. The analyses catalog classifies all the registered analysis services with respect to a specific software analysis taxonomy. The *Software Analysis Broker* web service acts as the interface between the catalog and the users.

Specific ontologies are used to define and represent the data consumed and produced by the different analysis services, while upper ontologies define much more generic concepts common to several specific ontologies. Thus, they provide semantic links between them, which otherwise would remain decoupled.

In the following, we explain these constituents in greater detail.

Software Analysis Services

Our solution proposes software analyses to be available as web services. We decided to leverage this paradigm over other competing middleware technologies as it is a well known standard and it was devised to overcome some of the problems we also face and thus already offers many of the features we need, namely: language, platform and location independence and service composition.

Independence is achieved with the use of XML-based languages to describe the services (WSDL [31]) and a simple, lightweight communication protocol (SOAP [32]) intended for exchanging structured information, formatted into XML-based messages, in a decentralized, distributed environment, normally using HTTP/HTTPS. Composition and orchestration is provided by BPEL4WS (Business Process Execution Language for Web Services) [6], an XML-based language designed to enable task sharing for a distributed computing—even across multiple organizations—using a combination of Web services.

Moreover, because of these characteristics of loose coupling, published interfaces and a standard communication model, existing applications can expose their functionalities through web services without significant changes. The internal logic, the input and output formats used, the platform and language under which the original tool runs remain the same but are hidden behind the web service not being a burden for interoperability anymore, as it has been shown by many works as, for example, [33, 34, 35]. At last, with the use of semantical annotated web services, they can be seamlessly integrated with ontologies, whose usefulness and significance in our solution will be explained later.

Software Analyses Catalog and Taxonomy

With web services research groups and software companies throughout the world can easily share, use and combine different analyses across organizational, geographical, platform and language boundaries through the Net. But

these services alone are not enough; in order to be effectively used they need to be kept track of and classified in some sort of registry. This is why we created the Analyses Catalog, which is used to store and classify all the registered analysis services so that any user can automatically discover analysis services she is interested in, invoke them and then fetch the results. To do that, a clear and univoque classification is essential. Based on the existing software analysis techniques we developed a specific software analysis taxonomy to systematically classify the existing and future services. This taxonomy divides the possible analyses into three main categories based on what aspect of a software system they focus on:

- the development process,
- the underlying models, or
- the actual source code.

Software development analyses are further divided into those targeting the development history (extraction, prediction and analysis of source code changes and bugs), its underlying process (its dynamics and metrics, as the ones defined by Lorenz *et al.* [36] and Nagappan *et al.* [17]) and the teams involved in it (their dynamics and metrics), as shown in Figure 2.

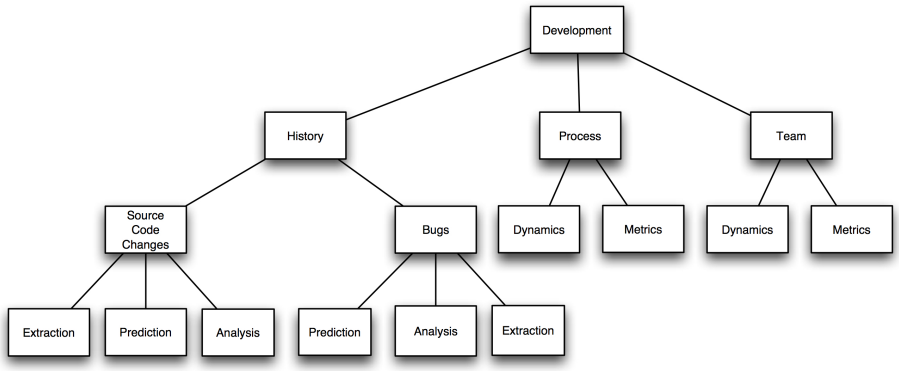

Fig. 2: A view of the software development analysis branch

Model analyses are further divided into those targeting the extraction, either dynamic or static, of specific behavioral and structural model representations (UML, FAMIX, call graphs, Rigi, etc.) and those computing differences between two models, usually of two versions of the same system. Figure 3 gives a condensed view of this part of the classification.

Code analyses, being the oldest and thus most studied topic of this taxonomy, are further divided into many other categories, as for example those checking code well-formedness, its correctness and its quality. For example, the code quality category is then further divided into subcategories dealing

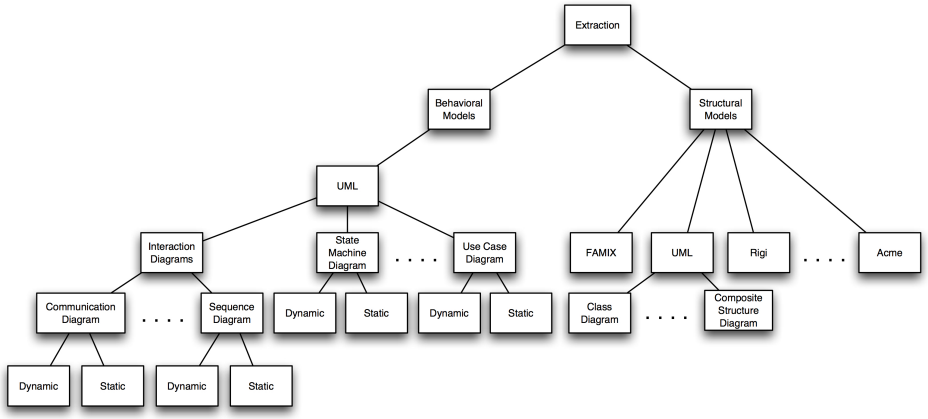

Fig. 3: A compact view of the software model extraction branch

with code security, conciseness, performance and design. This last category contains, among the others, extractors and analyzers of design metrics, as defined by Lanza *et al.* [37] and Lorenz *et al.* [36], and code-smells, as defined by Fowler *et al.* [38]. We will not go further into details due to space limitations and as it is beyond the scope of this chapter. But we decided to a least show the part about code design quality as these analyses are essential in the field of software evolution analysis to study whether and how the quality of the system under examination evolved.

This proposed taxonomy is obviously not the only one possible and by no means complete. But the proposed categories are reasonable enough and make sense, in particular from the perspective of a user who wants to find some particular analyses without struggling with many and sometimes obscure categorizations but at the same time wants them to be expressive and meaningful. Since, to our knowledge, the literature lacks any preexisting taxonomies of this kind, we structured it mainly using the currently existing approaches as a blueprint and so that they would "fit" reasonably well into that, but, as in any classification there are always individuals that do not clearly fit in any category or fit in more than one.

Ontologies: the need for semantic descriptions in software analysis

WSDL specifies a standard way to describe the interfaces of a web services, the structure of their input and output and how to invoke them at a syntactic level. However, there is no way to know what analysis a service actually offers. Each specific service would then still structure its results according to its specific format and following its own meta-model. Thus, the integration and combination of results would still be at most possible with cumbersome man-

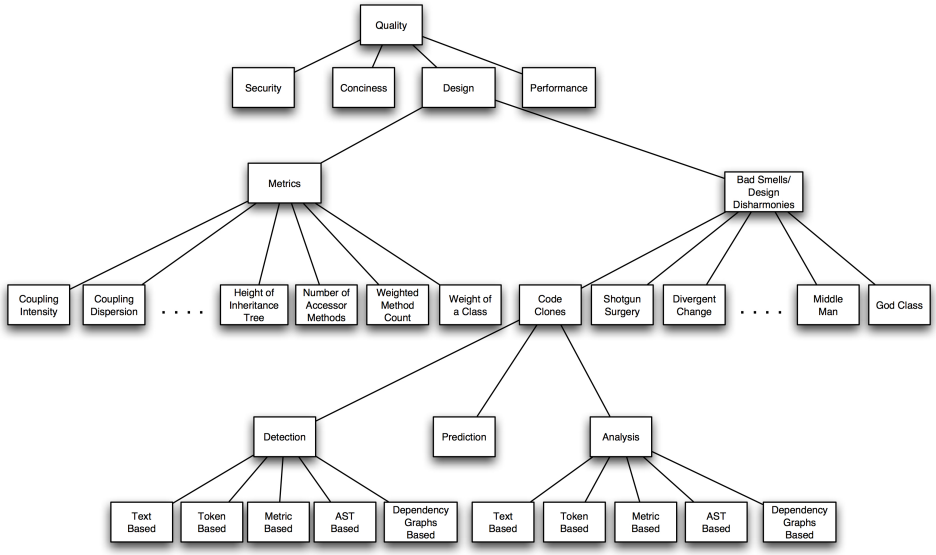

Fig. 4: A condensed view of the software design quality branch

ual ad-hoc data preparation and transformation. A common exchange format providing a rigorous, univoque syntax and semantic of data is indispensable.

Several researchers have pushed for common interchange formats such as GXL (Graph eXchange Language) [39] or XMI [40], but their efforts have remained largely unheard. The MSR (Mining Software Repositories) community is striving for integration especially in their Mining Challenge track, but it is limited to the application of the analysis tools on the same case studies. Moreover, the existing exchange formats focus only on the syntax of data, but do not address its semantic at all.

A promising alternative is to use ontologies, in particular OWL, to represent both results and input data. First it gives us a sound and well known data format to use and the ability to share that data between different types of computers using different types of operating system and application languages, as it is written in XML. Second, the properties related to its ontological nature make it really stand out from all the other already existing solutions: (1) heterogenous domain ontologies can be semantically "linked" to each other by means of one or more upper ontology, which describe general concepts across a wide range of domains. In this way it is possible to reach interoperability between a large number of ontologies accessible "under" some upper ontology. In terms of software analysis services, it means that results from the most disparate type could be automatically combined given that they share some common concepts; (2) with the OWL Description Logic foundation it is possible to perform automatic reasoning and derive additional knowledge;

(3) we can use a powerful query language such as SPARQL or its extension iSPARQL [41], that uses similarity operators to query for similar entities; and (4) in contrast to XML and XQuery [42] that operate on the structure of the data, OWL treats data based on its semantics. This allows for an extension of the data model with no backwards compatibility problems with existing tools.

Moreover, thanks to the recently introduced Semantic Annotations for WSDL and XML Schema [43], web services and ontologies can be effectively integrated together to create *semantic web services*, which are extremely valuable to us as crucial parts of our approach. Semantic annotations can be attached to any part of a web service definition, adding semantic meaning to it, as it is shown in Figure 5 (the semantic annotation are bold and circled). The example highlights the reasons why this approach is really useful for our purposes. First the service itself can be declared to represent a particular concept of an ontology, in our case a specific analysis category; in the example, a CVS release history data extractor. Second, its inputs and outputs can be declared of being concepts of specific ontologies and thus have a specific semantic meaning. In the example of Figure 5, since the service itself offers CVS release history, the output is then declared of representing a CVS history, as defined in a specific ontology (we will talk about that more in details in Section 4). In this way we know precisely what the service returns and what it means. Moreover, with all this information we can then easily check, for every new service being registered in our analyses catalog, whether it supports inputs and provide results conforming to ontologies specific to the analysis it is declared to implement (e.g. every service offering CVS extraction has to return a CVS history).

Software Analysis Broker web service

The *Software Analysis Broker* acts as a "layer" between the catalog and the users through which they can query, update, manage the catalog (namely register, update and unregister analysis services). They can even expand the taxonomy, as new types of analyses that were not yet classified, or some modification to the already existing classification, could come up in the future. More precisely, the *Software Analysis Broker* can be queried to get the content of the analyses catalog (in other words, the registered analyses) and if one or more specific analyses have been performed on some project. We decided to offer just these two functionalities because those two pieces of information are everything a user might want to know in this context. Furthermore, any additional information can then be fetched from a combination of them. Those two queries are offered through a web service interface and the results formatted into a standardized machine readable format, more precisely OWL. In this way tools of any kind can (semi)-automatically fetch the analyses they need to then call them without any human intervention. However, this makes the results hardly readable by humans. So we chose to let the *Software Analysis*

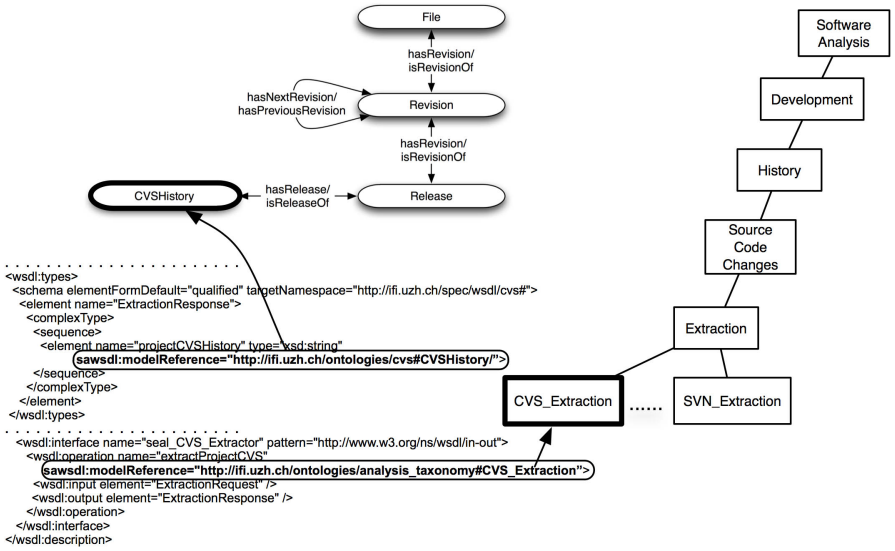

Fig. 5: An example of a semantical annotated web service definition

Broker be queried in the same way also through a website, which will format and present the results in a much more understandable form for human users. Therefore we will show the *Software Analysis Broker* functionalities through its website interface, keeping in mind that everything we will show can also be done by calling directly the web service. Figure 6 shows the initial view that is presented to the user. The user can either navigate through the catalog or query it. With the navigation option he/she can get an idea of the analysis taxonomy structure or see what the analyses being offered are (note that the classification used for the navigation is the same we presented in the previous section). With queries more specific information for the successive invocation of the services can be gathered.

Figure 7 shows what the *Software Analysis Broker* returns when queried for the currently registered analyses which is essentially the current instance of the catalog. So for every service is reported the name, the address through which it can be invoked and the type of analysis offered. Knowing the latter gives the user all the information on the service input and output. In fact, as we explained in the previous section, every analysis type is associated with ontologies to which the input and output of every service offering it must conform. Thus with this query it is possible to know what analyses can be performed and gather all the information needed to then call the desired ones. So it will be used when a user or a tool, given a project, wants to conduct some analysis and has to know who is currently offering it.
Figure 8 shows what the *Software Analysis Broker* returns when queried to find out if one or more types of analysis were performed on some specified

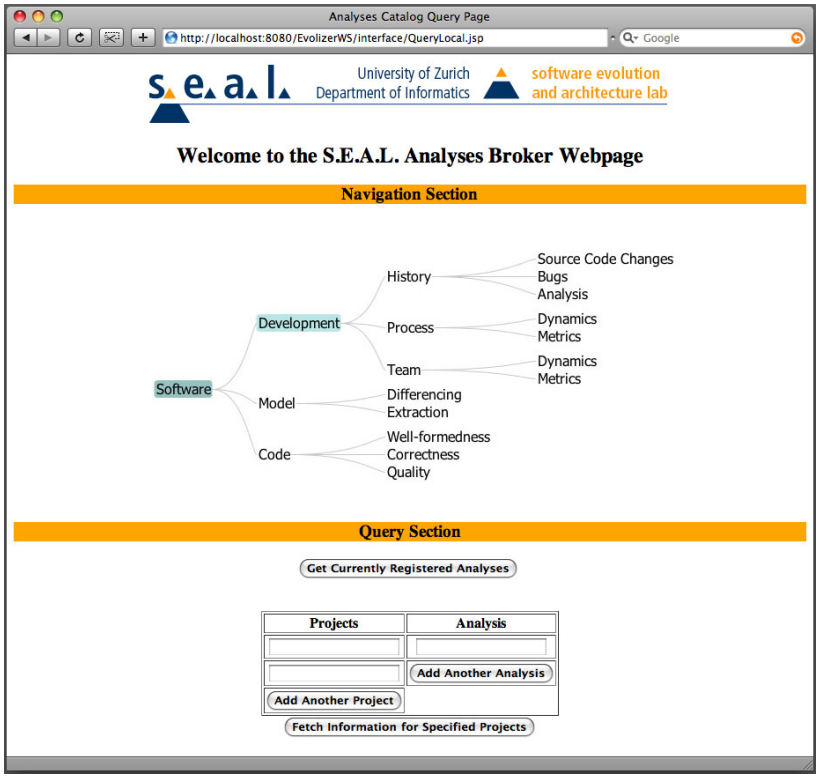

Fig. 6: The initial page of the *Software Analysis Broker* website

projects. Note that for all the projects is displayed whether or not every single requested analysis has been already performed, without explicitly showing what is the actual service that didit. In fact, as long as it is performed, it does not really matter who performed the analysis since, as we explained before, all the services offering it will comply to a common output both syntactically and semantically. Nevertheless the address of the actual service offering the analysis is simply hidden by the HTML representation behind the "check" symbol. So it can be immediately invoked to get the available data without having to query the *Software Analysis Broker* for any other information.
All this information is useful to see what data about a project is already available to then fetch it or trigger the analysis to produce it. Furthermore, it can be handy for tools and users that need case study data from existing projects to then run their own analysis. For example a tool extracting some newly defined software project metrics might need CVS history data of software projects for case studies and proofs of concept for validation. So, instead of finding suitable projects and extracting their CVS data by itself, it could take advantage of the previous analyses and thus just fetch the data that has

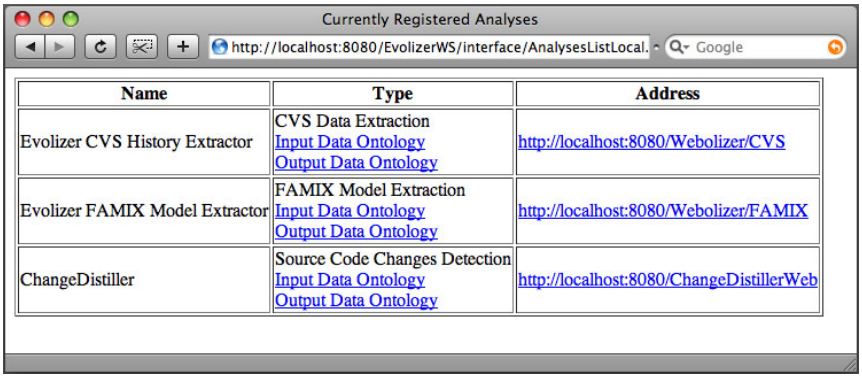

Fig. 7: The registered analysis services

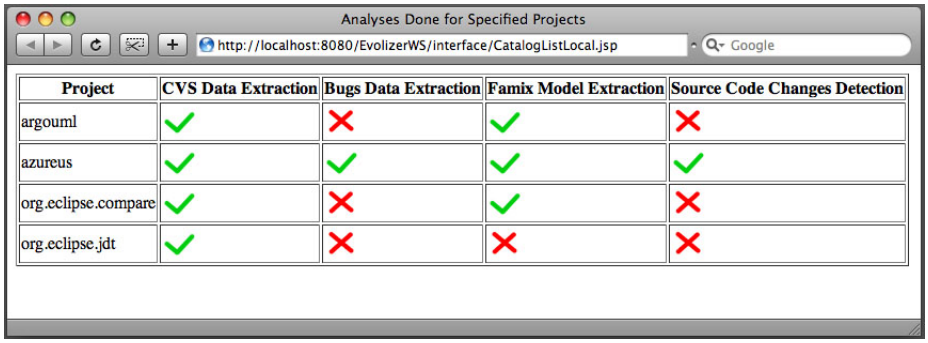

Fig. 8: *Software Analysis Broker* list of analyses and projects

already been extracted by the registered services offering CVS data extraction.

Moreover, with the *Software Analysis Broker* web service, we can also add more complex functionalities, such as service composition, on top of the analyses catalog which would allow us to fully exploit our platform. For example, if a user wants a series of analyses performed on a project, he/she coud either search the catalog for the desired analyses, compose them through a web interface and then execute them or he/she could let the *Software Analysis Broker* take care of finding, composing, executing them (for example with BPEL) and then just get the final results once the whole process is done.

4 Software analysis services at work

In this section we present an excerpt of software analysis services that we have implemented and show how they can be orchestrated to solve the task outlined in our analysis scenario. All services exploit techniques and tools that have been implemented by our group comprising the CVS importer, Bugzilla importer, and FAMIX parser. For each service we show its semantical annotated definition and the ontologies of the needed input and generated output data.

4.1 CVS History Extractor service

```
. . . . . .
<wsdl:types>
 <schema elementFormDefault="qualified" targetNamespace="http://ifi.uzh.ch/spec/wsdl/cvs#">
  <element name="ExtractionRequest">
   <complexType>
    <sequence>
     <element name="repositoryUrl" type="xsd:string"/>
    </sequence>
   </complexType>
  </element>
  <element name="ExtractionResponse">
   <complexType>
    <sequence>
1    <element name="projectCVSHistory" type="xsd:string"
          sawsdl:modelReference="http://ifi.uzh.ch/ontologies/cvs#CVSHistory/">
    </sequence>
   </complexType>
  </element>
</wsdl:types>
. . . . . .
  <wsdl:interface name="seal_CVS_Extractor" pattern="http://www.w3.org/ns/wsdl/in-out">
2   <wsdl:operation name="extractProjectCVS"
       sawsdl:modelReference="http://ifi.uzh.ch/ontologies/analysis_taxonomy#CVS_Extraction">
     <wsdl:input element="ExtractionRequest" />
     <wsdl:output element="ExtractionResponse" />
    </wsdl:operation>
  </wsdl:interface>
</wsdl:description>
```

Fig. 9: Excerpt of the WSDL definition of the CVS Importer service

This service extracts the versioning information comprising release, revision, and commit information from a CVS repository. Figure 9 shows the definition of the web service. The service belongs to the "CVS_Extraction" category of our taxonomy, as it is declared by the WSDL element framed by box number 2. As such it needs an URL as input that specifies the location of the CVS repository. When running it connects to the repository, obtains and processes the CVS information and outputs the resulting data model in the format specified by the CVSHistory ontology (see WSDL element marked with 1).

The core concepts of the CVSHistory ontology are depicted in Figure 10. In addition to the directory structure, the importer obtains, for each file, all its revision information and corresponding modification reports. They basically contain the information on who changed when/which source file and how many lines have been inserted/deleted. That data is stored as RDF triples and a reference is provided to the user for accessing it. The reference can be queried from the *Software Analysis Broker* so that the processing of particular CVS repositories needs to be done only once. Any subsequent request can use the saved triple store.

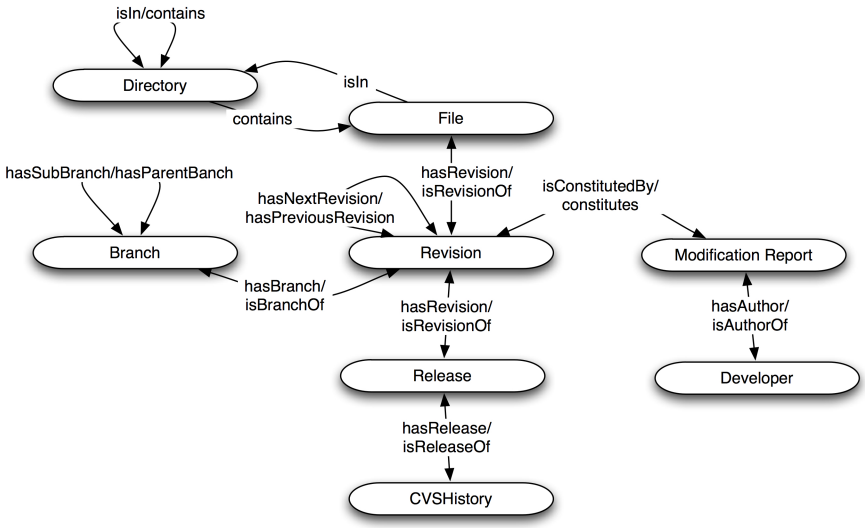

Fig. 10: High-level view of the CVS history ontology

4.2 Bugzilla Extractor service

This service extracts problem reports and change requests from a Bugzilla repository. The WSDL definition is shown in Figure 11. The service belongs to the "Bugzilla_Extraction" category which is a subcategory of generic bug extraction services (see element marked 4 in the WSDL definition). Similar to the CVS Importer service, it needs a string denoting the URL of the location of the Bugzilla repository as input. When run the service accesses the Bugzilla repository to derive the problem reports and change requests in XML format, parses them and stores the result as RDF triples. The triples conform to the ontology shown in Figure 12 and referenced by the element marked with 2.

Optionally, the client of the service can provide a reference to an already imported CVS model (see element marked 1). When the reference is given,

```
. . . . . .
<wsdl:types>
 <schema elementFormDefault="qualified" targetNamespace="http://ifi.uzh.ch/spec/wsdl/bugzilla#">
  <element name="ExtractionRequest">
   <complexType>
    <sequence>
     <element name="bugzillaUrl" type="xsd:string"/>
1    <element name="projectCVSHistory" type="xsd:string"
          sawsdl:modelReference="http://ifi.uzh.ch/ontologies/cvs#CVSHistory"/>
     <element name="cvs_to_bugs_linkage_heuristic" type="xsd:string"/>
    </sequence>
   </complexType>
  </element>
  <element name="ExtractionResponse">
   <complexType>
    <sequence>
2    <element name="projectCVSHistory" type="xsd:string"
          sawsdl:modelReference="http://ifi.uzh.ch/ontologies/bugzilla#IssueHistory"/>
3    <element name="bugs_CVS_Links" type="xsd:string"
          sawsdl:modelReference="http://ifi.uzh.ch/ontologies/cvs_bugzilla#CVS_Bugzilla_Links"/>
    </sequence>
   </complexType>
  </element>
</wsdl:types>
. . . . . .
  <wsdl:interface name="seal_Issue_Tracking_Extractor" pattern="http://www.w3.org/ns/wsdl/in-out">
4   <wsdl:operation name="extractProjectBugzilla"
       sawsdl:modelReference="http://ifi.uzh.ch/ontologies/analysis_taxonomy#Bugzilla_Extraction">
     <wsdl:input element="ExtractionRequest" />
     <wsdl:output element="ExtractionResponse" />
    </wsdl:operation>
  </wsdl:interface>
</wsdl:description>
```

Fig. 11: Excerpt of the WSDL definition of the Bugzilla Importer service

the service runs a procedure that establishes the links between CVS Revision and Issue entities. As no standard to report a bug fix or a reference to a bug in the CVS commits is enforced (usually the developers add the related bug reference number in the commit message), in order to effectively reconstruct those links, some heuristics are needed. Our service can be configured to use several of them, from very simple and trivial ones to the ones proposed by Sliwerski *et al.* [44] and by Mockus *et al.* [45] which are more structured. That inferred linking data is structured as an instance of the simple ontology shown in Figure 13, which basically associates each extracted Bugzilla issue to all its related CVS revisions of the CVS History that was passed as input and viceversa. Also these links are stored as RDF triples with the ontology referenced by the element marked 3.

4.3 Famix Model Extractor service

Like the Bugzilla Extractor also this service, as shown by the WSDL element framed by box number 1 in Figure 14, requires as input a project CVS history data structured as a "CVSHistory" concept. Given that information it then fetches, for each project release, its source code and parses it to get the related Famix model and transforms it into a specific Famix ontology, shown

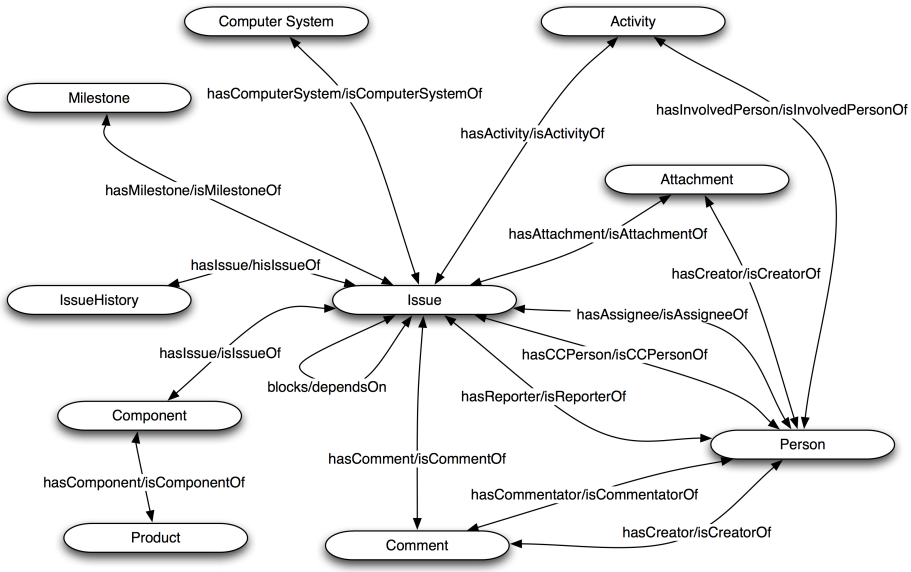

Fig. 12: High-level view of the issue tracking history ontology

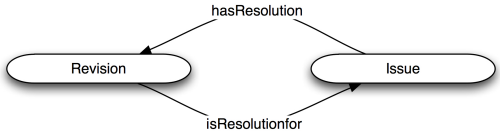

Fig. 13: High-level view of the Issue Tracking-Cvs links ontology

in Figure 15. That data is then returned as the output of the service (see element marked 2) and it is also stored as RDF triples. The CVS history this service requires as input is not only used to know and fetch all the releases of the project of interest, but it is used also to create links between the CVS history and the Famix Model created. This information is represented using the ontology shown in Figure 16. The links keep track of all the CVS revisions in which a Famix Class (which represents the generic OO class concept) was modified and vice versa. From the Class entity all the remaining information on its related Famix Model can be easily fetched and in the same way, from its linked Revision, all the associated CVS information can be gathered. As before, the data is returned as part of the service output, as shown by the WSDL element marked 3, and stored as RDF triples.

```
. . . . . .
<wsdl:types>
 <schema elementFormDefault="qualified" targetNamespace="http://ifi.uzh.ch/spec/wsdl/famix#">
  <element name="ExtractionRequest">
   <complexType>
    <sequence>
1    <element name="projectCVSHistory" type="xsd:string"
          sawsdl:modelReference="http://ifi.uzh.ch/ontologies/cvs#CVSHistory"/>
    </sequence>
   </complexType>
  </element>
  <element name="ExtractionResponse">
   <complexType>
    <sequence>
2    <element name="projectCVSModel" type="xsd:string" maxOccurs="unbounded"
          sawsdl:modelReference="http://ifi.uzh.ch/ontologies/famix#FamixModel"/>
3    <element name="CVS_Famix_Links" type="xsd:string"
          sawsdl:modelReference="http://ifi.uzh.ch/ontologies/cvs_famix#CVS_Famix_Links"/>
    </sequence>
    </sequence>
   </complexType>
  </element>
 </wsdl:types>
. . . . . .
  <wsdl:interface name="seal_Famix_Model_Extractor" pattern="http://www.w3.org/ns/wsdl/in-out">
4  <wsdl:operation name="extractProjectFamix"
     sawsdl:modelReference="http://ifi.uzh.ch/ontologies/analysis_taxonomy#FAMIX_Extraction">
     <wsdl:input element="ExtractionRequest" />
     <wsdl:output element="ExtractionResponse" />
   </wsdl:operation>
  </wsdl:interface>
</wsdl:description>
```

Fig. 14: Excerpt of the WSDL definition of the Famix Model Extractor service

4.4 An interoperability scenario

If we come back to the analysis scenario we presented in Section 1, the services we introduced before are more or less the ones that Alice, Charlie and Bob agreed to offer after their meeting. So, what happens when they are integrated in our *Software Analysis Broker* platform and a user, in our case the fourth person of the scenario, Jane, comes into play?

As a first step she needs to check what is available on the catalog and in particular whether services offering the required analyses exist and are registered. In order to do that she queries the *Software Analysis Broker* to see whether some CVS history, Bugzilla history and Famix extraction services are available. Figure 17 shows a schematic snapshot of the taxonomy showing in particular under which category those required services are classified. Note that the category they belong to is exactly the same that is declared in their service definitions we illustrate above. Then, by fetching the semantically annotated WSDL file for each of them, which we already showed in Figure 9, Figure 11 and Figure 14, Jane can learn what input data needs to be supplied with and what is their expected output. So in this case she finds out that in order to get all the data she wants, first she needs to call the CVS History Extractor and then once the results are ready provide them to the

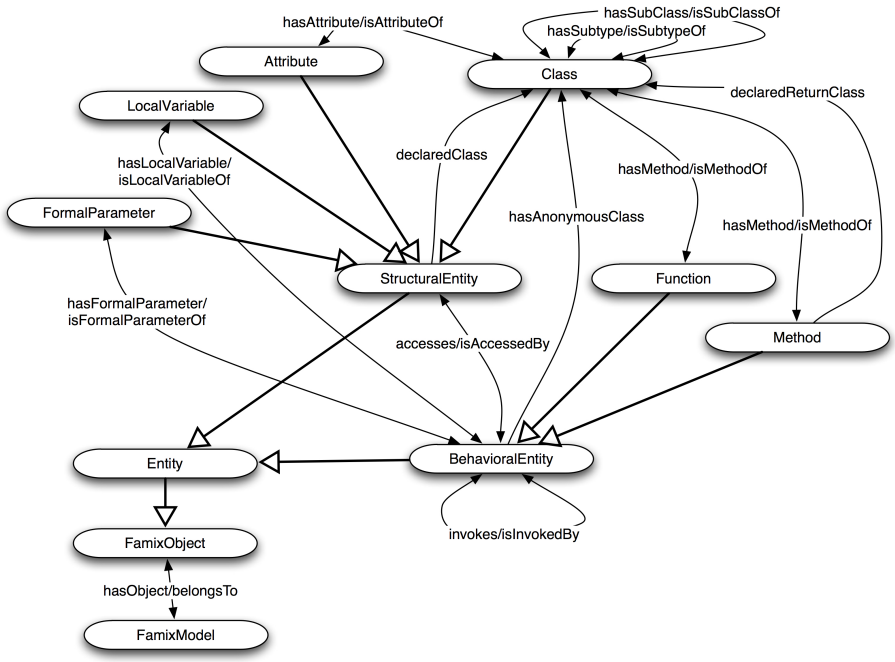

Fig. 15: High-level condensed view of the Famix model ontology

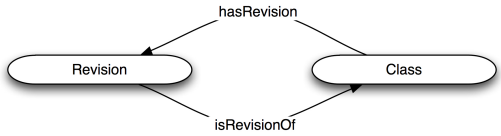

Fig. 16: High-level view of the Famix-Cvs Links ontology

Bugzilla Extractor and the Famix Model Extractor so that they can carry out their analyses, as they require CVS History data to perform their analyses. She can do this all by herself by getting the reference to the services and invoking them in the required order and with the right inputs. Or, even better, she instructs the *Software Analysis Broker* to compose those three services into a BPEL workflow and have it run on a BPEL engine, which will actually take care of the whole flow passing the data from one service to another, as we already mentioned in Section 3.1. In this way she only needs to specify at a high level how to compose and run the different services without having to deal and know BPEL itself. In this case the whole execution of the services is more or less automated. This is extremely useful for the combination of time consuming analyses, either for the type of the analysis itself or for the analyzed system size. In Jane's case, the body of knowledge that

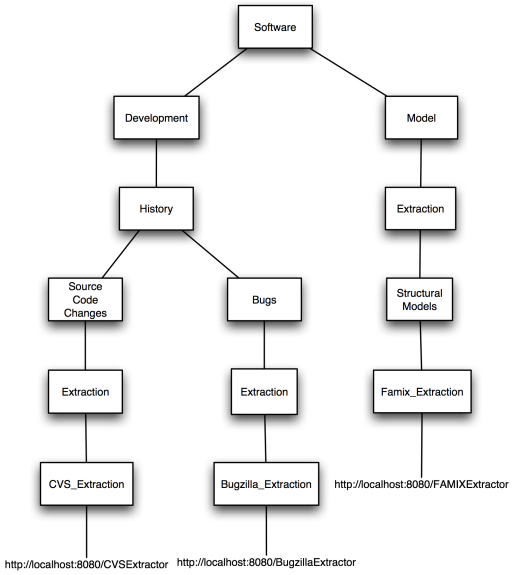

Fig. 17: The categories the existing services belong to

she's eventually able to get is shown in Figure 18. Note how, thanks to the links between the CVS and Bugzilla data and the CVS and Famix data, all the ontologies are linked. She can then proceed to examine that huge amount of data and fetch all the information needed for her analysis using SPARQL queries or any other approach of her choice. By querying the Famix data of every project release she can extract all the source code metrics she needs to spot all the possible code smells and then get all "smelly" classes. Due to the links established between the Famix and the CVS data she can get for any of those classes all their revisions and from there, with the links between the Issue Tracking and the CVS data, get all the issues associated to them. With that data she can then run her own analysis on the relation between code smells and bugs: to see whether code smells caused the emergence of bugs and/or bug fixing reduces the amount of code smells.
The same job, without our platform would have required the installation and configuration of at least three different tools, the ad-hoc transformation to and from the different formats used by them and the manual linking of that different data (or developing an ad-hoc tool to do that). On the other hand, with our solution, it boils down to the invocation of just a few web services and the running of some SPARQL queries, with no tedious and error prone data preparation, code modification, etc.

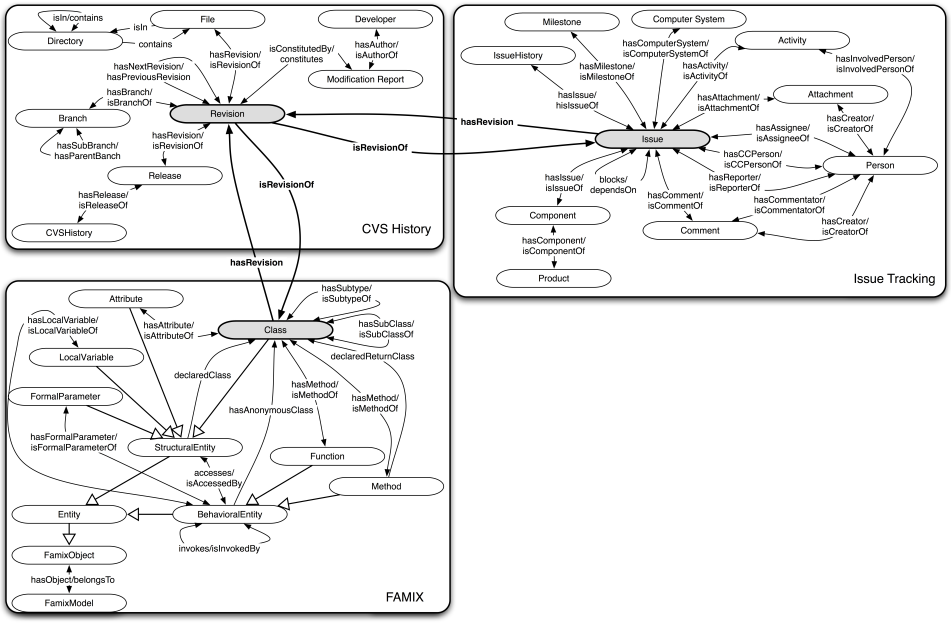

Fig. 18: Overall view

5 Conclusions

The combination and integration of different software analysis tools is a challenging problem when we need to gain a deeper insight into a software system's evolution. For every required analysis a specialized tool, with its own explicit or implicit meta-model dictating how to represent the input and the output, has to be installed, configured and executed. Even if different analyses of the same kind exist, the only way to compare them is to do it manually.

Our approach aims to solve that problem with *a combination of ontologies and web services for software analysis services*. Using web services to expose the functionalities offered by the analysis tools gives us independence from platform, language and location. Further we can apply well-known mechanisms of service composition and orchestration (*e.g.,* BPEL4WS) of several analysis services. OWL ontologies specific to distinct types of analyses allow us to have standard formats to define and represent the data consumed and produced by the analysis services, which can then be integrated with each other based on semantic "links". These links are provided by generic, upper ontologies. With semantically annotated web services, we can formalize for each service the actual ontological concepts and its input and output by just adding a few annotations in the service definition. Moreover, it is then possible to support (semi)-automatic composition of services based on the semantic of their input and output. And at last, but not at least, due to OWL's powerful

query language SPARQL and its Description Logic foundation, data can be extracted and additional knowledge can be inferred with existing tools.

Finally, we are convinced that by allowing disparate analysis tools to collaborate with each other and share their information via a service platform can be highly beneficial. Not only it will enhance and speed up the work of a software engineer by giving him/her access to a big amount of information without the need to install several tools and to cope with many output formats, but it would also promote the uncovering of new meaningful and interesting metrics deriving from the most diverse types of analysis that can finally "talk" to each other.

The work we have presented is a major endeavor and as such still work in progress. However, everything shown here is part of already existing prototypes that we developed. The *Software Analysis Broker* and the services, along with all their related ontologies, have been realized and extended into full-fledged web services so that they can be used by outside users. The (semi)-automatic composition of services using semantics has not been implemented yet but we have started to address it. This is because first we wanted to have an initial version of the whole infrastructure up and running with just a few services registered. This should point out the possible problems and issues to guide subsequent improvements and to show the feasibility and usefulness of the approach. In this way also external users could start using our own analyses to see how it works, grasp its potentials and maybe integrate their own tools. In fact, since it is a platform for distributed and collaborative software analysis, we would like to have other research groups share their analysis approaches through our platform, and thereby making it really valuable.

References

1. Fischer, M., Pinzger, M., Gall, H.: Populating a Release History Database from Version Control and Bug Tracking Systems. Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003 (2003) 23–32
2. Demeyer, S., Tichelaar, S., Steyaert, P.: FAMIX 2.0 - The FAMOOS Information Exchange Model, Technical Report. IEEE Transactions on Software Engineering (1999)
3. Fluri, B., Würsch, M., Pinzger, M., Gall, H.C.: Change Distilling–Tree Differencing for Fine-Grained Source Code Change Extraction. IEEE Transactions on Software Engineering **33**(11) (November 2007) 725–743
4. Fluri, B., Gall, H.C.: Classifying Change Types for Qualifying Change Couplings. Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006) (June 2006) 35–45
5. Schwarz, B.: Sna-cockpit. Master's thesis, University of Zurich, Department of Informatics (April 2007)
6. Jordan, D., Evdemon, J.: Web Services Business Process Execution Language Version 2.0. OASIS Standard. (April 2007)

7. Jackson, D., Rinard, M.: Software analysis: a roadmap. Proceedings of the Conference on The Future of Software Engineering at ICSE 2000 (2000) 133–145
8. Draheim, D., Pekacki, L.: Process-Centric Analytical Processing of Version Control Data. Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE 2003) (2003) 131
9. Zou, L., Godfrey, M.: Detecting Merging and Splitting Using Origin Analysis. Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003) (2003) 146154
10. Kim, S., Pan, K., Whitehead, E.: When functions change their names: automatic detection of origin relationships. Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005) (2005) 23–32
11. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An Empirical Study of Code Clones Genealogies. Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2005) (2005) 23–32
12. Zimmermann, T., Weissgerber, P., Diehl, S., Zeller, A.: Mining Version History to Guide Software Changes. Proceedings of the 26th International Conference on Software Engineering (ICSE 2004) (2004) 563572
13. Ying, A.T.T., Murphy, G.C., Ng, R., Chu-Carroll, M.C.: Predicting Source Code Changes by Mining Change History. IEEE Transactions on Software Engineering **30**(9) (2004) 574–586
14. Livshits, B., Zimmermann, T.: Dynamine: Finding Common Error Patterns by Mining Software Revision Histories. Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2008) (2005) 296–305
15. Čubranić, D., Murphy, G.C.: Hipikat: Recommending Pertinent Software Development Artifacts. Proceedings of the 25th International Conference on Software Engineering (ICSE 2003) (2003) 408–418
16. Gall, H., Jazayeri, M., Krajewski, J.: CVS Release History Data for Detecting Logical Couplings. Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE 2003) (2003) 13–23
17. Nagappan, N., Ball, T.: Use of Relative Code Churn Measures to Predict System Defect Density. Proceedings of the 27th International Conference on Software Engineering (ICSE 2005) (2005) 284–292
18. Hassan, A., Holt, R.: The Top Ten List: Dynamic Fault Prediction. Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005) (2005)
19. Kim, S., Zimmermann, T., Jr., E.W., Zeller, A.: Predicting Faults from Cached History. Proceedings of the 29th international conference on Software Engineering (ICSE 2007) (2007) 489–498
20. Sliwerski, J., Zimmermann, T., Zeller, A.: HATARI. Raising Risk Awareness. Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2005) (2005) 107–110
21. des Rivières, J., Wiegand, J.: Eclipse: A platform for integrating development tools. IBM Systems Journal **43**(2) (2004) 371–383
22. Weiss, C., Premraj, R., Zimmermann, T., Zeller, A.: How Long will it Take to Fix This Bug? Proceedings of the 4th International Workshop on Mining Software Repositories (MSR 2007) (2007)

23. Čubranić, D., Murphy, G.C.: Automatic Bug Triage Using Text Categorization. Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE) (2004)
24. Anvik, J., Hiew, L., Murphy, G.C.: Who should fix this bug? Proceedings of the 28th International Conference on Software Engineering (ICSE 2006) (2006) 361– 370
25. Mockus, A., Herbsleb, J.: Expertise Browser: a quantitative approach to identifying expertise. Proceedings of the 24th International Conference on Software Engineering (ICSE 2002) (2002) 503–512
26. Gîrba, T., Kuhn, A., Seeberger, M., Ducasse, S.: How Developers Drive Software Evolution. Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE 2005) (2005) 113–122
27. Hyland-Wood, D., Carrington, D., Kaplan, S.: Toward a Software Maintenance Methodology using Semantic Web Techniques. Proceedings of the 2nd International IEEE Workshop on Software Evolvability at IEEE International Conference on Software Maintenance (ICSM 2006) (2006) 23–30
28. Happel, H., Korthaus, A., Seedorf, S., Tomczyk, P.: KOntoR: An Ontology-enabled Approach to Software Reuse. Proceedings of the 18th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE 2006) (2006)
29. Kiefer, C., Bernstein, A., Tappolet, J.: Mining Software Repositories with iSPARQL and a Software Evolution Ontology. Proceedings of the 4th International Workshop on Mining Software Repositories (MSR 2007) (2007)
30. Jin, D., Cordy, J.R.: Ontology-Based Software Analysis and Reengineering Tool Integration: the OASIS Service-Sharing Methodology. Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005) (2005) 613–616
31. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1. W3C Note. (March 2001)
32. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Nielsen, H., Karmarkar, A., Lafon, Y.: SOAP Version 1.2. W3C Recommendation. (April 2007)
33. Canfora, G., Fasolino, A., Frattolillo, G., Tramontana, P.: Migrating interactive legacy systems to web services. Proceedings of the 10th European Conference on Software Maintenance and Reengineering (2006)
34. Sneed, H.M., Sneed, S.: Creating web services from legacy host programs. Proceedings of the 5th IEEE International Workshop on Web Site Evolution (2003) 59–65
35. Sneed, H.: Integrating legacy software into a service oriented architecture. Proceedings of the 10th European Conference on Software Maintenance and Reengineering (March 2006)
36. Lorenz, M., Kidd, J.: Object-oriented software metrics: a practical guide. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1994)
37. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
38. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
39. Winter, A., Kullbach, B., Riediger, V.: An Overview of the GXL Graph Exchange Language. In: Revised Lectures on Software Visualization, International Seminar Dagstuhl Castle, London, UK, Springer-Verlag (2001) 324–336

40. OMG: MOF 2.0/XMI Mapping, Version 2.1.1. (January 2007)
41. Kiefer, C., Bernstein, A., Stocker, M.: The Fundamentals of iSPARQL - A Virtual Triple Approach For Similarity-Based Semantic Web Tasks. Proceedings of the 6th International Semantic Web Conference (ISWC) (2007) 295–309
42. Boag, S., Chamberlin, D., Fernández, M., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language. (January 2007)
43. Farrell, J., Lausen, H.: Semantic Annotations for WSDL and XML Schema, W3C Recommendation. (August 2007)
44. Sliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? Proceedings of the 2th International Workshop on Mining Software Repositories (MSR 2005) (2005)
45. Mockus, A., Votta, L.G.: Identifying Reasons for Software Changes using Historic Databases. Proceeding of the 16th International Conference on Software Maintenance (ICSM 2000) (October 2000) 120–130